The eXtensible Data Model and Format for Interdisciplinary Computing

Mr. Jerry A. Clarke
U.S. Army Research Laboratory
Aberdeen Proving Ground , Maryland 21005-5067
clarke@arl.army.mil

Dr. Raju Namburu
U.S. Army Research Laboratory
Aberdeen Proving Ground , Maryland 21005-5067
raju@arl.army.mil

Abstract

The Interdisciplinary Computing Environment (ICE), a CHSSI portfolio project, has defined a common, active data model and format to help combine various High Performance Computing (**HPC**) codes and tools into a single user system. Known as the eXtensible Data model and Format (***XDMF***), it provides computational engines with the tools necessary to exist in a modern computing environment with minimal modification. Instead of imposing a new programming paradigm on HPC codes, XDMF uses the existing concept of file I/O for distributed coordination. XDMF incorporates Network Distributed Global Memory (***NDGM***), Hierarchical Data Format version 5 (***HDF5***), and eXtensible Markup Language (***XML***) to provide a flexible yet efficient data exchange mechanism.

Naturally, the flexibility and functionality of the system sacrifices some performance when compared to a "hard-wired" solution. A balance between functionality and performance must be reached that allows for reusable tools that perform their function with acceptable efficiency. In addition, to be truly useful, existing HPC programs must be able to take advantage of the system without overly burdensome modification. To gauge the actual costs in runtime of this flexible system, two different codes were outfitted for use. First Paradyn, an MPI based finite element code, was tested in a distributed memory environment. Next CTH, an MPI based finite volume code was run in a shared memory environment. Due to the different nature of the data layout in these codes, they are good representatives of the ends of our performance spectrum.

Introduction

Modern High Performance Computing (HPC) facilities are commonly a collection of heterogeneous systems, each selected for its particular strength. Logically, software systems can benefit from combining the strengths of various hardware systems, under a single operating environment, to provide feature rich end-user environments. In addition, software systems can also benefit from combining components from different disciplines in a single system. For example, providing interactive runtime scientific visualization to a running HPC code allows the user to verify setup and monitor progress during execution. Computation can proceed on the large scalable system, while the user visually analyses the data on a high-end graphics workstation.

Developing an environment where distributed and parallel software components, written in various programming languages, can be easily assembled into a complex system is a difficult task. In a high performance computing environment, the task is even more difficult since the volume of data being processed is enormous. It is beneficial to minimize data movement in order to improve overall system performance.

Existing low level facilities like sockets, Remote Procedure Call (RPC), Message Passing Interface (MPI), and Parallel Virtual Machine (PVM), alone are insufficient to build complex, reusable distributed computing components and applications. In addition to the extensive bookkeeping necessary to coordinate messages, they lack the standard facilities to describe the *meaning* of the data in addition to its' *values*. For example, it is impossible to know whether an array of floating point values describes the X,Y,Z values of a computational grid or computed vector values without some type of additional information. Additional layers, provided by higher level facilities and environments, address this issue.

Many higher level efforts attempt to provide a generalized solution for distributed applications. Some of the most notable efforts involve Meta-computing, Distributed Object Brokers, and Software Buses. Serious Meta-Computing efforts attempt to address the complete distributed computing issue in its entirety. Facilities for user authentication, job submission, resource allocation, and task coordination are provided in an effort to provide seamless access to a potentially enormous computational grid. These systems typically provide some facility for individual components to communicate in addition to standard low level methods. Meta-computing environments like Globus [1] and Legion [2] may potentially provide the necessary framework, but properly deploying these systems requires significant site-wide coordination of queuing systems, system software, accounting, and security policies. These systems are effectively distributed operating systems and will require more time to attain the stability and acceptance necessary for site-wide deployment in production HPC environments. In addition, they provide no inherent facility for describing data meaning since they must support *every* type of HPC code.

Environments implemented entirely of user level code require no privileged access and simplify both accounting and security issues. While limited in functionality by definition, these environments provide sufficient capability for the implementation of robust end-user systems constructed from verified and efficient components in various disciplines. If each of these components were to be developed from scratch, one might take advantage of object oriented, distributed systems based on the Common Object Request Broker Architecture (CORBA) [3]. However, imposing this type of object oriented architecture on an existing HPC code designed for scalable performance requires significant effort and could adversely affect the code's performance.

Other approaches like the Polylith [4] software bus provide a standard method for components to provide *services* to *clients* upon request. As its name implies, the software bus approach is a flexible mechanism to interconnect diverse software components in a procedural, rather than an object-orientated architecture. As opposed to a targeted solution, this approach is a generalized one focusing on the communication and data transformation of independent interconnected modules. Polylith, as well as other systems like Darwin [5] define a "Module Interconnection Language" to describe the structure of the distributed system. The Olan Configuration Language [6] found existing object request brokers and software bus "module interconnection languages" insufficient and extended the approaches to convey the dynamic aspects of applications.

While these systems have met with varying degrees of success, none can be considered a "standard" for constructing distributed applications from existing components. These systems attempt to define a "top-down" structure for

defining and implementing the operation of the entire software system. In fact, this “top-down” approach is common to meta-computing and CORBA systems. Perhaps a more expedient approach for sharing data is a “bottom-up” one of defining a common data facility to which HPC codes could efficiently read and write values in a standardized method similar to a restart or dump file. Such a “bottom-up” solution to this issue would be immediately useful for transferring data between applications like HPC simulation codes and visualization post-processors. Eventually, if simultaneously available to running applications, such a solution would lend itself to co-processing and computational steering systems. Even this proves to be a non-trivial undertaking. Defining a method for accessing enormous amounts of data and describing its content is difficult when one considers the diverse data organization requirements and performance issues of modern parallel codes. Several efforts attempt to define common data models, formats, and sharing mechanisms for HPC applications.

While it is unlikely that a single data model and format could serve all HPC applications adequately, clusters of similar applications can agree on data exchange mechanisms. For example, the HDF-EOS [7] effort has made significant progress toward providing a common layer of data access for earth science data collection. The Department of Energy ASCI program has significant effort to provide a common data model and format, based on HDF5, for DOE simulation codes and tools. Other significant efforts like the KeLP (Kernel Lattice Parallel) [8] system from the University of California, San Diego, and the Active Data Repository [9] from the University of Maryland , have succeeded in providing flexible data access for running applications.

To realize the benefits of these systems, HPC codes and tools must use custom APIs to access data. For example, an application would not use the HDF5 API to access HDF-EOS data, rather it would utilize the HDF-EOS API. This forces the application to be aware of, or adopt, the underlying data model. Full support for more than one of these systems in an HPC simulation is rare at best. Legacy codes or codes with limited development manpower, may find it difficult to implement new data access concepts not designed into the original code. In addition, current systems with a clearly defined data model tend to store that information (metedata attibutes) via the same mechanism used to implement the data format. This is not always optimal and makes it difficult to extend the data model or embed additional information outside of the formal model.

Simply stated, the problem is that current systems either don’t provide a sufficient data model or make it difficult to separate from the data format. A parallel HPC code, for example, should not be constrained by the data model required by a generalized visualization system; the code should be able to efficiently update values and continue computation. This is the focus of the XDMF solution. By separating the data model from format, the necessary order can be *associated* with the raw values to give them meaning while not constraining the HPC code. The code only needs to be concerned with producing the proper format, leaving the complexity of the data model to other tools. In addition, the data model is extensible, so particular features not part of the original design can be added as needed.

Origins in Runtime Visualization

At the U.S. Army Laboratory, we have developed a system called the “Distributed Interactive Computing Environment” (DICE) and are using it in a production environment with structural mechanics, fluid dynamics, and computational chemistry HPC codes. DICE provides flexible yet efficient mega-components for graphical user interface, data organization, data exchange, and scientific visualization. Originally designed to provide distributed runtime visualization capability to several computational fluid dynamics codes, DICE has become a flexible way to provide distributed capabilities to a variety of HPC codes in different computational technology areas.

XDMF is the successor to the DICE data model. XDMF categorizes data in two logically separate flavors: Heavy Data and Light Data. Heavy data are potentially enormous data structures that describe data *values*. Light data conveys data *meaning* and generally contains information describing the heavy data. Light data tends to be small and portable, while access to the heavy data should be minimized for performance reasons. XDMF keeps these two types of data logically, and (when advantageous) physically, separate. As we will see, this provides an enormous amount of functionality and flexibility to distributed software components.

Instead of imposing a new paradigm on HPC codes, XDMF uses the existing concept of file I/O for coordination. Since virtually every major HPC code has input and output sections, it is straightforward to add additional calls that mimic standard file I/O to transfer calculated values and control information to the XDMF system. However, instead of resulting in secondary storage I/O, these calls result in distributed interprocess communication.

Network Distributed Global Memory (NDGM), is a facility like Distributed Shared Memory (DSM) that provides efficient access to a virtual, contiguous buffer via a client/server architecture. NDGM servers manage a portion of a buffer that physically resides in system shared memory, a disk file, or in its local address space. Clients access this buffer via puts(), gets(), and vector operations by specifying a virtual start address of the desired data. Locating sections of requested buffers and the actual transfer is handled by the NDGM client system. In addition, NDGM provides semaphores and barriers to coordinate the activity of parallel/distributed applications.

While NDGM is not a true DSM system it provides several advantages when designing an efficient data exchange mechanism. NDGM does not automatically map and unmap pages of memory from an application. Rather, the application itself is responsible for transferring data to NDGM via a read/write interface.

While this might appear to be a major inconvenience, it greatly reduces the amount of communication. In addition, NDGM takes full advantage of operating system shared memory facilities when a server and client reside on the same machine. This allows remote clients to maintain full access to the data buffer while local clients incur minimal overhead. Support for the "Generic Security Services API" (GSS-API) [10] provides the necessary security features that are typically required in an HPC environment. NDGM by itself was used to create an efficient parallel version of an existing CFD code without significant restructuring of the original code [11].

The original DICE data model was targeted at CFD codes that used large, multi-block structured grids, so it was possible to implement simple, existing file formats directly in the NDGM buffer with minimal effort. However, the addition of codes from other disciplines made the need for a more comprehensive data organization obvious. The Hierarchical Data Format Version 4 (HDF) from NCSA was chosen to implement a self-describing organization over NDGM that provided sufficient facilities for accessing sub-sections of data. The low level HDF I/O routines were modified to provide transparent access to NDGM via the HDF API. HDF datasets could now exist on disk, in NDGM, or both. This proved convenient since codes could use the NDGM facility for interprocess communication and/or write to secondary storage during batch execution by simply modifying the target HDF filename.

Upon attempting to integrate the system with more production codes, limitations of the current system became apparent:

- HDF4 used a fixed 32-bit offset mechanism resulting in a 2GB limit.
- There was little in the way of data description above the rank and dimension of arrays. For example, other than name, there was no way of telling XYZ position data from calculated scalars.
- Other than NDGM access on pre-initialized HDF "files", there was no parallel I/O facility.
- Even tools that required information about possibly invariant quantities, like rank and dimension, needed to access the potentially remote binary data in HDF. This required development in system programming languages like C and C++ when scripting languages like Tcl and Python were more appropriate.

Several of these limitations could be easily resolved by migrating to the new HDF5 format being developed primarily for NASA's Earth Observing System HDF-EOS project, and the DOE ASCI Scientific Data Management (SDM) comprehensive data model and format effort. However, by revisiting some of the basic concepts of the system, we have significantly improved flexibility while maintaining efficiency.

XDMF: The Next Step

We recently implemented a data model and format, which can best be described as a distributed data hub, called the eXtensible Data Model and Format (**XDMF**). It is currently being integrated into several major HPC codes. The first major feature of XDMF is that the *Light Data* (data about the data and small amounts

of values) is logically, and potentially, physically separate from the *Heavy Data* (large arrays). The second major feature is the support of Network Distributed Global Memory as a virtual file driver under NCSA's Hierarchical Data Format Version 5 (**HDF5**).

XDMF provides a targeted data model and format as well as a facility for sharing the data in a distributed environment during runtime. Through the use of NDGM, codes and tools can synchronize their activities at a coarse grain level to provide a complex end-user application consisting of individually simple components. Access to XDMF is provided via system programming languages like C, C++, and FORTRAN, scripting languages like Tcl/Tk, Python, and Java.

The primary advantage of XDMF is interoperability. Tools can be quickly designed that perform a particular task very well. If they use XDMF, these tools can then be reassembled in a variety of configurations to accomplish different goals. New HPC codes or visualization tools need only provide XDMF access to use any of the other tools. XDMF is both a data model and format; information about data values and "how they are to be used", are both made available.

It is important to note that XDMF is intended to augment, not replace traditional parallel computing facilities like MPI, PVM, and OpenMP. In practice, existing components that use MPI are outfitted with a small amount of additional code to update XDMF. The code's efficiency is maintained via traditional, well accepted means, while the overall distributed application is serviced by XDMF.

Using XDMF and other components, a total end user application system can be assembled. This system provides access to pre-processing, code setup, runtime support and post processing.

As shown in Figure 1, HDF5 is used to provide the NDGM buffer with structure. HDF has been enhanced to support data in NDGM via the "virtual file driver" interface. No HDF code needs to be modified to support this functionality; the NDGM "driver" can simply be used with the current version of HDF5. With the addition of this driver, HDF datasets can reside on disk, in NDGM, or in both. This is particularly useful when data has both a static and dynamic component. For example, a static grid may reside on disk while the updating solution resides in NDGM. HDF provides a consistent interface for structured and unstructured data as well as a grouping structure. It allows the storage of character, integer, and floating point values in a portable fashion by providing conversion to various host dependent formats. In addition, since the layout of the data is tightly related to access efficiency, HDF5 provides multi-dimensional data access facilities as part of the interface.

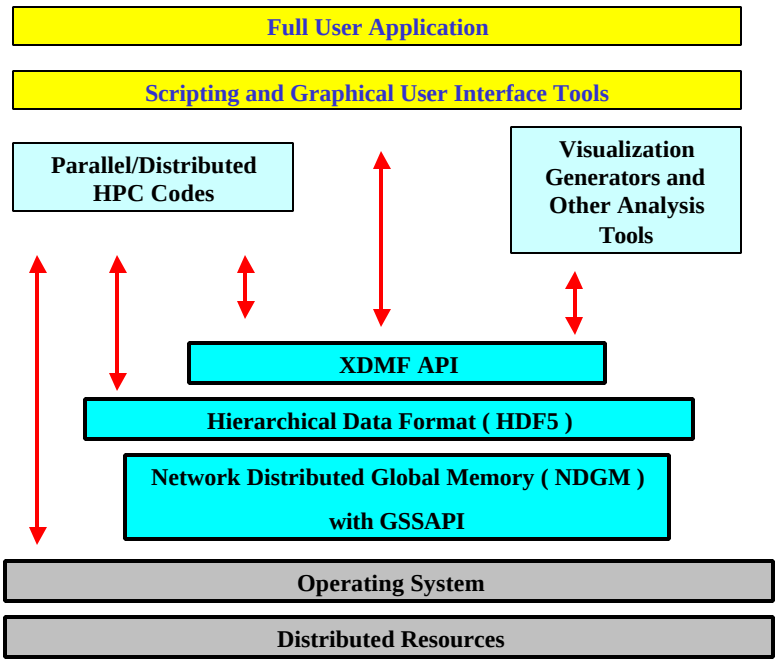

Figure 1. Access to XDMF.

XDMF supports a variety of character, integer and floating point data types. These individual data types are then organized into XDMF *Arrays*. Arrays are self-allocating, multi-dimensional, data structures that have methods to set and get the number of elements, safely access individual element values, and to directly manipulate the underlying data pointer for maximum efficiency. Several arithmetic operators have been overloaded and an additional "expression" facility has been supplied to allow operations on entire arrays. A sub-region of the array may be described by Hyperslab or Index. A Hyperslab specifies a Start, Stride, and Count in each dimension while an Index describes parametric indexes into a dataset. Commonly, Hyperslabs are used to subsection structured datasets while Indexes are used to subsection unstructured datasets.

XDMF arrays are stored externally in HDF5 "files". HDF provides a "Virtual File Driver" layer to allow HDF5 "files" to physically reside in things other than standard disk files. XDMF provides an NDGM driver in addition to the available Global Access to Secondary Storage (GASS : Globus System) and CORE (In-Memory) drivers provided with HDF5. HDF5 also provides a compression facility so that data can be compressed and decompressed as it migrates to/from physical storage. XDMF arrays inside an HDF5 file must be fully qualified for access. This is done by providing the Domain, File, and Pathname of the dataset. This is passed as a colon separated string (i.e. NDGM:Myfile.h5:/Geometry/XYZdata). Accepted domains are : FILE, CORE, GASS, and NDGM.

To effectively describe the data in a flexible manner, some type of data model is needed. Since it is anticipated that any model will need to be augmented by the application in order to include all of the necessary information; flexibility and extensibility are key elements to the data model. The intent of the data model component is to provide a way to easily describe data content as opposed to data value.

In concept, this is quite similar to a Web page where the content of the information is separate from the display of the information (i.e. color is independent of a word's meaning). To assist in the free exchange of data on the internet, the "World Wide Web Consortium" (W3C) [12], an organization whose members include AOL, IBM, Microsoft, Oracle, Sun, and other major corporations, proposed a standard known as "eXtensible Markup Language" (XML). XML is pervasive on the Web and is supported by a vast amount of tools, both free and commercial.

While HDF provides an attribute facility capable of storing light data such as units and dimension names, we feel a better choice is to take advantage of the recent emergence of XML. Although primarily targeted at web based applications, XML provides a standard way to store and structure application specific data. There is already an impressive availability of tools for parsing XML and converting it to internal data structures. The base data model including information like grid topology, scalar names, and physical data location, is stored using XML. By utilizing the functionality of XML, and logically separating the light data from the HPC heavy data, a myriad of tools can be built with high level scripting languages and web tools that allow intelligent queries of enormous datasets without causing massive amounts of I/O activity. In addition, this makes it easy for separate components to view the same data values differently. For example, one component's structured grid may be viewed by another component as a collection of hexahedra. We believe that the ability to physically separate the light data from the heavy data provides an enormous benefit.

The data model in XDMF is stored in XML. This provides the knowledge of what is represented by the Heavy data. In this model, HPC data is viewed as a hierarchy of Domains. A Domain may contain one or more sub-domains but must contain at least one Grid. A Grid is the basic representation of both the geometric and computed/measured values. A Grid is considered to be a group of elements with homogeneous Topology and their associated values. If there is more than one type of topology, they are represented in separate Grids. In addition to the topology of the Grid, Geometry, specifying the X, Y, and Z positions of the Grid is required. Finally, a Grid may have one or more Attributes. Attributes are used to store any other value associated with the grid and may be referenced to the Grid or to individual cells that comprise the Grid.

The XML may be passed as an argument, stored in an external file or communicated via a socket mechanism. For customization purposes, tools can also augment the standard content with XML "processing instructions". This is useful for attaching peer level information to the standard XML content without modifying the base specification.

The concept of separating the light data from the heavy data**,** as shown in Figure 2, is critical to the performance of this data model and format. HPC codes can read and write data in large, contiguous chunks that are natural to their internal data storage, to achieve optimal I/O performance. If codes were required to significantly re-arrange data prior to I/O operations, data locality, and thus performance, could be adversely affected, particularly on codes that attempt to make maximum use of memory cache. The complexity of the dataset is described in the light data portion which is small and transportable. For example, the light data might specify a topology of one million tetrahedra while the heavy data would contain the geometric XYZ values of the mesh and pressure values at the cell centers stored in large, contiguous arrays. This key feature will allow reusable tools to be built that do not put onerous requirements on HPC codes.

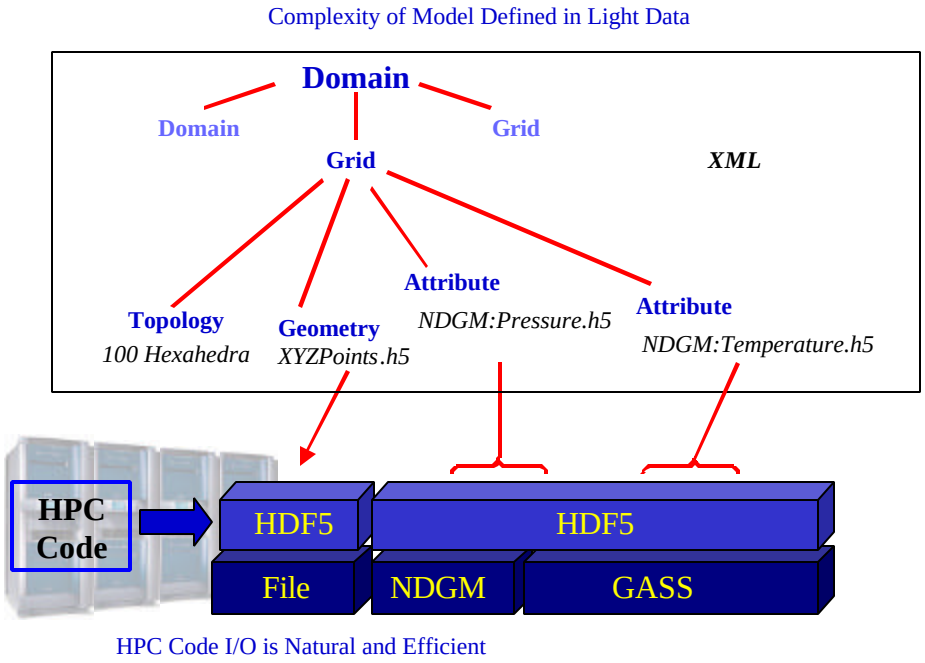

Figure 2. Separating Light and heavy data

XDMF : Programming Language Access

In addition to accessing data values via the HDF interface, XDMF contains an object-oriented C++ convenience interface to the XML light data and HDF5 heavy data. This convenience interface layer provides access to a subset of the HDF calls using reasonable default values for many of the parameters. It is also able to encapsulate and synchronize data, in memory and external (HDF) representations, to assist in the development of a more interactive set of data analysis tools. Components that benefit from an object oriented architecture can access data via this layer but its use is not required.

The computationally intensive components of a large system are generally developed using system-programming languages like C, C++, or FORTRAN. Once these computationally intensive components have been developed however, they may be "glued" together in a number of ways to provide the overall functionality required. Using a system-programming language for this task is tedious, time-consuming, and inflexible.

Scripting languages are specifically designed for this purpose. They tend to be "weakly-typed" so that the output of one component can easily be used as the input to another with little concern for the "type" of the data. While one pays in runtime efficiency for this flexibility, scripting languages are intended to call large chunks of functionality and not be used for fine grain control. By using "*The Simplified Wrapper and Interface Generator*" (SWIG) the XDMF functionality is made available to languages like Tcl, Python, and Java. We have used this interface to develop a simple visualization system based on the "Visualization Toolkit" (vtk). In addition, we have developed a data reader for the commercial visualization package EnSight. The combination of support for vtk and EnSight services the vast majority of user visualization needs.

XDMF: Component Coordination

Providing a common distributed data model and format alone is insufficient for building distributed applications. Individual components must be able to easily coordinate their activity in order to avoid polling and race conditions. Without this facility we have little more than a network file system.

NDGM provides barriers and semaphores for this purpose. Semaphores are locks, obtained by a client, explicitly released by the client or automatically released when the client exits. Barriers are used to coordinate the actions of a group of clients. A barrier is first initialized to a given value. Then as clients "check into" the barrier, the value is decremented. The client's activity is suspended until the value reaches zero. NDGM also provides a "barrier audit" facility where a client can "check into" a barrier without effecting it's value. This is useful for transient components, like visualization tools, to update their information in a read-only manner.

These coordination facilities are made available to the upper layers of XDMF. When a parallel HPC code begins to update values, each computational node can optionally wait in a barrier while all other nodes complete their updates. Also the update can be suspended until a semaphore has been released by a controlling component (typically a runtime visualization monitor).

Functionality vs. Performance

Naturally, the flexibility and functionality of the system sacrifices some performance when compared to a "hard-wired" solution. A balance between functionality and performance must be reached that allows for reusable tools that perform their function with acceptable efficiency. In addition, to be truly useful, existing HPC programs must be able to take advantage of the system without overly burdensome modification. To gauge the actual runtime costs in runtime of this flexible system, two different codes were outfitted for use. First Paradyn, an MPI based finite element code, was tested in a distributed memory environment. Next CTH, an MPI based finite volume code was run in a shared memory environment. Due to the different nature of the data layout in these codes, they are good representatives of the ends of our performance spectrum.

ParaDyn, from Lawrence Livermore National Laboratory, is a parallel version of the widely used, finite element based structural dynamics program Dyna3D . ParaDyn, like many currently used HPC codes, is written primarily in FORTRAN and uses MPI to achieve parallelism. We felt that adding runtime visualization capability to ParaDyn, would demonstrate the steps required to integrate existing components and also result in a useful distributed application at ARL.

ParaDyn, like many HPC simulation codes, follows this basic execution:

- Read in computational grid and input parameters from the file system
- Initialize internal variables
- Iterate over the core physics routines of the code until final solution is reached, periodically writing intermediate solutions to the file system
- Write final solution, cleanup, and exit

The additional XDMF calls map well into this execution flow. Since the code is mainly FORTRAN and XDMF access is accomplished via C++, FORTAN wrapper functions are needed to encapsulate the required functionality. For example, we write a new FORTRAN subroutine PARAINIT( ), called when ParaDyn initializes its internal variables, to initialize the necessary XDMF C++ objects and store their addresses in static variables. When the nodes need to update XDMF, they have access to the appropriate C++ objects.

The internal structure of ParaDyn is complex enough to place it beyond the scope of our discussion. Suffice it to say that internal variables are accessed through an internal database API. For simplicity, let us assume that there exists such FORTRAN subroutines as PDGETXYZ(), PDGETNODEVAR(), and PDGETCELLVAR() to return XYZ location and scalar values. We add a subroutine call to the main ParaDyn loop to call a new PARACHECK() subroutine every iteration. This is the where the majority of the XDMF functionality is accessed.

PARACHECK() checks for new requests and for previous requests that are due. For example, a previous request might have arranged for data to be updated every 5E-04 seconds of computational time or every 10 iterations. All requests are made via XML and processed via the internal XML parser supplied with the convenience layer. Data values are retrieved via the appropriate ParaDyn database API routines, and written via an HDF5 object also

supplied in the convenience layer. The HDF5 object handles any remote access via NDGM internally and automatically handles errors like the specified NDGM server becoming inaccessible, which is a common occurrence. For example, it is sometimes desirable for the user to temporarily start an NDGM server, request an update, then remove the NDGM server when convinced the code is behaving correctly. Since a single run may take many wall clock hours, these checks may be initiated from different locations; NDGM may need to move. Pseudo code for PARACHECK() follows :

```
PARACHECK ( integer Iteration, real SimulationTime )
        Check for new requests
        If Update is Required {
                Parse XML Request
                For each node in the request {
                        Switch on Request {
                                Case XYZ : data = PDGETXYZ()
                                Case Node : data = PDGETNODEVAR()
                                Case Cell : data = PDGETCELLVAR()
                                Default : Log Request Error
                                }
                        Map local subdomain of this variable to global space
                        Write data to HDF5 in global space
                }
        If this is a scalar run or this is MPI node 0 {
                        Signal completion of update via NDGM barrier
                        }
        }
```

The XML to describe the data being written by ParaDyn gives the raw data meaning. For example, Paradyn writes arrays of nodal position and connectivity, the following section of XML describes how those arrays define a hexahedral mesh.

```
<Grid Name="Solid Elements">
        <Topology
                Type="Hexahedra"
                NumberOfElements="110520">
                <DataStructure
                        Dimensions=”110520 8”
                        DataType=”Int”
                        Format=”HDF”>
                NDGM:Blocks.h5:/Connections/Solid
                </DataStructure>
        </Topology>
        <Geometry Type=”XYZ”>
                <DataStructure
                        Dimensions=”179685 3”
                        DataType=”Float”
                        Precision=”8”
                        Format=”HDF”>
                NDGM:Blocks.h5:/node/Values/Position
                </DataStructure>
        </Geometry>
</Grid>
```

As a benchmark, a ParaDyn simulation was run of a concrete block wall being loaded by a blast. The visualization below shows the initial loading of the wall represented by the colors on a mesh at the original concrete block locations. At that point in the simulation, the initial loading, gravity, and contact with other blocks effect the block's displacement. The lowest row of concrete block is fixed to the ground and not allowed to move. The simulation is small enough (about 110,000 hexahedral elements) to complete in a reasonable amount of time for benchmarking purposes. In fact, the relatively small amount of data being sent to the NDGM buffer and the noncontiguous nature of the unstructured mesh magnifies communication latency effects. In addition, in order to get a "worst case" idea of the overhead involved, we ran the problem on the IBM NH-2 (375 MHz Power3, the jobs were submitted in a manner to distribute the work across different SMP nodes), with the NDGM buffer on one node. In this scenario, all of the MPI nodes must funnel their runtime data to one designated "collection" node that holds the HDF5 data.

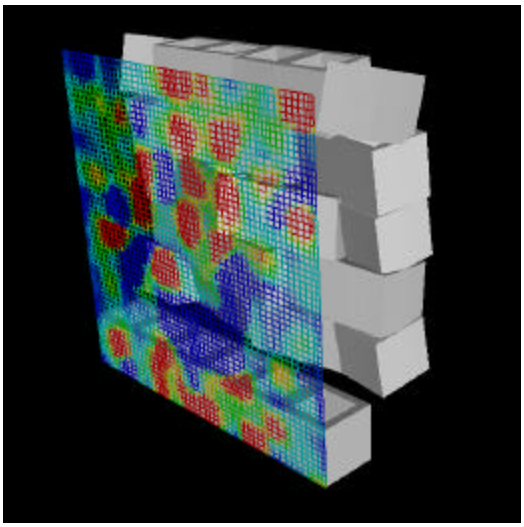

Figure 3. ParaDyn Simulation

The NDGM implementation uses TCP/IP sockets as a communication mechanism. To get an idea of what to expect, we ran the UNIX ttcp (Test TCP) program, with various buffer sizes, between nodes on the IBM. With minor variations we regularly measured an average performance of about 68 MB/sec between 2 nodes for buffers over 25KB. Smaller buffers significantly reduced performance. Buffers of 2KB only see 9MB/sec while buffers of 200 bytes can see less than 1MB/sec. A test of the low level NDGM calls revealed an average of about 61 MB/sec over transfer sizes above 25KB between 2 nodes. This test writes contiguous sections of the NDGM buffer without calling any HDF5 routines. This rate seems to stay constant with small numbers of nodes; i.e. with two client NDGM nodes accessing a third server node, each client sees about 30MB/sec.

A similar test to write then read back the buffer did not perform nearly as well. Each read requires the client to issue a request to the server, then collect responses. This test consistently yielded transfer rates between 18-20 MB/sec at best between two nodes. Smaller buffers yielded lower rates.

The problem time of 3E-02 seconds resulted in 2107 iterations in ParaDyn and an average timestep of 1.4E-05 seconds. Via the input file we specified an update frequency of 5E-04 seconds which resulted in an update every 35 iterations. Each update modified about 23MB of HDF5 data. Including the update at iteration 0, ParaDyn transferred about 1.4 GB of data to HDF5 over the entire runtime of the problem. This varies slightly for different numbers of nodes.

All of these timings were run on non-dedicated machines and networks. It is important to note that we are in no way attempting to study the performance or scalability of ParaDyn itself, or the platforms on which it ran. We wish only to demonstrate how to estimate the performance costs one can expect when implementing this new functionality. For each set of timings "effective throughput" is calculated. This is simply the total data written (1.4 GB) divided by the additional runtime required for the updates.

Table 1. Total Runtime on IBM-NH2

Number of Nodes	Total Time No Updates	Total Time with Updates	Effective Throughput
2	921 sec	1074 sec	9.15 MB/sec
4	462 sec	576 sec	12.28 MB/sec
6	308 sec	414 sec	13.21 MB/sec
8	239 sec	373 sec	10.45 MB/sec

While the performance was acceptable for runtime visualization purposes, we were a bit disappointed with the effective throughput. Adding some counters to the NDGM server revealed that it serviced 55,627 requests from an 8 node ParaDyn run. And while most of those requests were data writes, only about 20% of the writes were over 10KB. This is primarily due to the noncontiguous nature of unstructured data but the overhead of implementing a file structure via HDF5 adds additional requests for file control information.

To give an idea of how we perform in a heterogeneous distributed environment, we ran the same problem on an SGI Origin2000 (300 MHz R12000 Processors) with the NDGM buffer on a Sun E10000 (400MHz Ultrasparc II) over the Gigabit Ethernet interface. Running ttcp with 100KB buffers shows ~26MB/sec for this route. The low-level NDGM test showed ~23MB/sec effective throughput.

Table 2. Total Runtime between SGI Origin200 and Sun E10000

Number of Nodes	Total Time No Updates	Total Time with Updates	Effective Throughput
2	1653 sec	1761 sec	12.96 MB/sec
4	766 sec	871 sec	13.33 MB/sec
6	492 sec	593 sec	13.86 MB/sec
8	399 sec	495 sec	14.58 MB/sec

Using the Gigabit Ethernet connection, our effective throughput is about 13MB/sec. Tests over other interfaces yield similar numbers. This seems to indicate that local data access and communications latency are the main performance factors as opposed to data bandwidth. We will continue to investigate the lower levels of the system in order to better understand all of the pertinent issues in order to increase the effective throughput.

The same overall strategy has been used to outfit another code, CTH, for use with XDMF (Figure 4.). CTH is a heavily used, parallel, finite volume structural mechanics HPC code. It too has an internal database API to access variables. CTH, however, computes values on structured grids, so the data writes to the HDF5 file tend to be large contiguous blocks. This time, to see how this system performs under optimal conditions, we put the NDGM buffer on the same machine as the code. This results in the HDF5 access being kernel shared memory access. A simulation of a kinetic energy projectile impacting a moving armor plate was used for this test. A problem with a grid of 64 x 256 x 128 (2.097E6) cells was run on the SGI Origin2000 for 521 iterations (4E-06 seconds simulation time). This time we requested a data update every 10 iterations. Each data update is about 120MB so the total for the entire runtime is 6.36GB.

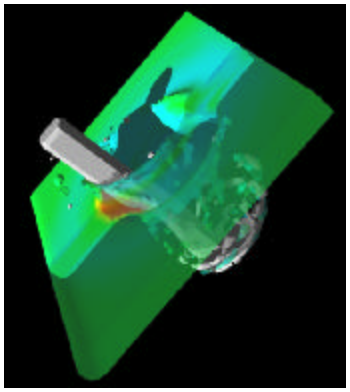

Figure 4. CTH Simulation

Table 3. Total Runtime on SGI Origin2000

Number of Nodes	Total Time No Updates	Total Time With Updates	Effective Throughput
8	6413 sec	6525 sec	56.8 MB/sec
16	4347 sec	4395 sec	133 MB/sec
32	2327 sec	2356 sec	219 MB/sec

Not surprisingly, the effective throughput of the structured grid code dumping data to shared memory is significantly better than the unstructured grid code dumping data to distributed memory or across the network. Large, contiguous data access results in less internal overhead and fewer messages or shared memory accesses. Since each processor updates the HDF5 data independently, the 32 processor runs, with an effective throughput of over 200MB/sec probably benefited by some of the HDF5 accesses overlapping computation of other processors. The simple lesson is that data layout is extremely important to performance.

With both ParaDyn and CTH, most of the interface code deals with accessing the internal database APIs. To provide a more straightforward example, we provide a complete code on our WEB site (www.arl.hpc.mil/ice) that has pre-processing, computation, and file I/O confined to a single FORTRAN source file. The code needed to interface this code to XDMF is then added in the previously mentioned fashion. This interface is about 150 lines of C++ code. Much of this interface is reusable for other applications; the main difference is the access of the HPC code's internal variables. This may result in a new convenience object to encapsulate the functionality thus reducing many interfaces to significantly less code.

In all cases, once the data is written to the HDF5 buffer it is ready to be visualized. Utilizing a graphical user interface, we provide access to common visualization techniques like isosurfaces and cutting planes through vtk "networks". The light data is used to initialize these networks before any heavy data is read from the HDF5 buffer. This flexibility allows the user to use the same visualization tools across a wide variety of HPC codes. For the more heavily used codes like ParaDyn and CTH, a Tcl/Tk interface is added to the environment for setting up code input thus providing an entire common runtime environment.

Conclusion

The eXtensible Data model and Format is a new approach to distributed computing. By mimicking the process of standard file I/O, XDMF adapts to the existing structure of many HPC codes. Consisting entirely of user level code, XDMF requires no site-wide deployment of privileged code or modification of current accounting or security policy. On this foundation, we have also provided the graphical user interface and visualization support necessary to develop an entire distributed environment for HPC codes.

Acknowledgment

The authors would like to acknowledge Ms. Jennifer Hare for her help implementing the system, Dr. Photios Papados for his help with ParaDyn, Mr. Stephen Schraml for his help with CTH, and the ARL MSRC for stable access to a variety of HPC platforms.

References

1. Foster, I., Antonio, J., “The Globus project: a status report”, proceedings of the Seventh heterogeneous Computing workshop, pp 4-18, march 1998

2. Grimshaw, A., Ferrari, A., Knabe, F., Humphrey, M., “Wide area computing: resource sharing on a large scale”, Computer , volume 32, issue 5, pp 29-37, May 1999

3. Object Management Group, “The common object request Broker: Architecture and Specification”, num. 91.12.1, December 1991

4. Purtilo, J.M., “The POLYLITH Software Bus”, ACM TOPLAS, Vol 16, Number 1, pp 151-174, Jan. 1994

5. Magee, J., Dulay, N., Kramer, J., “A Constructive Development Environment for Parallel and Distributed programs”, Proceedings of the International Workshop on Configurable Distributed Systems, Pittsburgh, March 1994

6. Bellissard L., Boyer, F., Riveill, M., Vion-Dury, J., “System Services for Distributed Application Configuration”, Proceedings of Fourth international conference on Configurable Distributed Systems. Pp 53-60, May 1998

7. Folk, M., McGrath, R., Yeager, N., “HDF: an update and future directions”, Proceedings of IEEE 1999 international Geoscience and Remote Sensing Symposium, Volume 1, pp 273-275, july 1999

8. Baden, S.B., Fink, S.J., “A programming methodology for dual-tier multicomputers “, IEEE Transactions on software engineering, Volume 26, Issue 3, pp 212-226, March 2000

9. Chialin Chang, Kurc, T., Sussman, A., Saltz, J., “ Optimizing retrieval and processing of multi-dimensional scientific datasets”, Proceedings of 14th international Parallel and Distributed Symposium, 2000. pp. 405-410, May 2000

10. Linn, J., RFC 1508, “Generic Security Service Application Program Interface”, September 1993

11. Clarke J., “Emulating Shared Memory to Simplify Distributed-Memory Programming”, IEEE Computational Science & Engineering, Vol 4, No. 1, pp 55-62, January-March 1997

12. “Document Object Model (DOM) Level 1 Specification.”, World wide Web Consortium, http://www/w3.org/TR/REC-DOM-Level-1